BladeDISC: Optimizing Dynamic Shape Machine Learning Workloads via Compiler Approach

ZHEN ZHENG*, Alibaba Group, China
ZAIFENG PAN†, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China and Alibaba Group, China
DALIN WANG†, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China and Alibaba Group, China
KAI ZHU, Alibaba Group, China
WENYI ZHAO, Alibaba Group, China
TIANYOU GUO, Alibaba Group, China
XIAFEI QIU, Alibaba Group, China
MINMIN SUN, Alibaba Group, China
JUNJIE BAI, Alibaba Group, China
FENG ZHANG, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China
XIAOYONG DU, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China
JIDONG ZHAI, Department of Computer Science and Technology, Tsinghua University, China
WEI LIN, Alibaba Group, China

Compiler optimization plays an increasingly important role to boost the performance of machine learning models for data processing and management. With increasingly complex data, the dynamic tensor shape phenomenon emerges for ML models. However, existing ML compilers either can only handle static shape models or expose a series of performance problems for both operator fusion optimization and code generation in dynamic shape scenes. This paper tackles the main challenges of dynamic shape optimization: the fusion optimization without shape value, and code generation supporting arbitrary shapes. To tackle the fundamental challenge of the absence of shape values, it systematically abstracts and excavates the shape information and designs a cross-level symbolic shape representation. With the insight that what fusion optimization relies

*Zhen Zheng is the corresponding author of this paper (james.zz@alibaba-inc.com).
†Work was done when Zaifeng and Dalin interned at Alibaba Group.

Authors' addresses: Zhen Zheng, Alibaba Group, China; Zaifeng Pan, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China and Alibaba Group, China; Dalin Wang, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China and Alibaba Group, China; Kai Zhu, Alibaba Group, China; Wenyi Zhao, Alibaba Group, China; Tianyou Guo, Alibaba Group, China; Xiafei Qiu, Alibaba Group, China; Minmin Sun, Alibaba Group, China; Junjie Bai, Alibaba Group, China; Feng Zhang, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China; Xiaoyong Du, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China; Jidong Zhai, Department of Computer Science and Technology, Tsinghua University, China; Wei Lin, Alibaba Group, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/9-ART206 $15.00
https://doi.org/10.1145/3617327

upon is tensor shape relationships between adjacent operators rather than exact shape values, it proposes the dynamic shape fusion approach based on shape information propagation. To generate code that adapts to arbitrary shapes efficiently, it proposes a compile-time and runtime combined code generation approach. Finally, it presents a complete optimization pipeline for dynamic shape models and implements an industrial-grade ML compiler, named BladeDISC. The extensive evaluation demonstrates that BladeDISC outperforms PyTorch, TorchScript, TVM, ONNX Runtime, XLA, Torch Inductor (dynamic shape), and TensorRT by up to 6.95×, 6.25×, 4.08×, 2.04×, 2.06×, 7.92×, and 4.16× (3.54×, 3.12×, 1.95×, 1.47×, 1.24×, 2.93×, and 1.46× on average) in terms of end-to-end inference speedup on the A10 and T4 GPU, respectively. BladeDISC's source code is publicly available at https://github.com/alibaba/BladeDISC.

CCS Concepts: • **Software and its engineering → Compilers**; • **Hardware → Emerging languages and compilers**; • **Computing methodologies → Machine learning**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Additional Key Words and Phrases: machine learning, tensor compiler, dynamic shape, operator fusion, code generation

ACM Reference Format:
Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, Xiaoyong Du, Jidong Zhai, and Wei Lin. 2023. BladeDISC: Optimizing Dynamic Shape Machine Learning Workloads via Compiler Approach. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 206 (September 2023), 29 pages. https://doi.org/10.1145/3617327

1 INTRODUCTION

Machine learning (ML) plays an increasingly important role in data management in recent years. How to optimize the ML system for efficient data management is becoming an important research topic [24, 27, 28, 37, 44, 53, 56]. Specifically, with the increasing diversity of data to process and manage, the need for the optimization of dynamic shape models[1] is growing [19, 20, 71]. In recent years, machine learning compilers [11, 14, 19, 30, 91] have become an indispensable component for model optimization and deployment. They accept the computation graph of a model and generate efficient code accordingly on target devices. However, existing optimizing compilers show limitations for dynamic shape models.

A common practice to deal with dynamic shape is to pad all dimensions of the input tensors (e.g., input and output sequence lengths) to the max value within each batch. However, there is an obstacle to applying conventional optimizing compiler techniques on the per-batch padding approach in many real-world production scenes. Most of the existing ML compilers [11, 14, 30, 91] are static shape oriented, which rely on dynamic recompilation (e.g., SystemML [29]) to deal with runtime-encountered shapes. Whenever a new shape comes (e.g., different batch sizes and sequence lengths after per-batch padding in different iterations), they recompile the graph, resulting in severe recompilation overhead and memory consumption of compilation caches for the encountered shapes. For example, we evaluate the overall latency for serving ten BERT-large inferences of different batch sizes and sequence lengths with TensorFlow, the XLA-optimized deployment spends 20× more of the execution time than naive TensorFlow due to the recompilation of every new shape. Note that a GPU kernel duration in a common ML program is usually (tens of) microseconds while the compilation of the kernel could be much longer.

As for just-in-time (JIT) compilation with caching using static shape compilers, it leads to a long warm-up process due to the frequent compilations and results in frequent jitter of service time for dynamic shape models. Note that the number of different shapes could be tens of thousands for

[1]Dynamic shape means the tensor shape could have varied values at runtime, which is common in real production. For example, the batch size and sequence length of language models [35, 67, 76] vary for different inputs, and the size of input images for computer vision models [36, 42] also varies.

some production tasks (e.g., $N_{batch} \times N_{seq-len}$ for NLP tasks, and $N_{batch} \times N_{img-size}$ for CV tasks). Moreover, the compilation cache consumes a large memory footprint. Note that the compilation cache is usually placed in memory, rather than on disk, to speed up the lookup and function load process. Finally, there are system expansion requirements in real business to serve the sudden increase in user requests. The newly expanded machines will suffer from the warming-up process again, as the in-memory compilation cache is hard to migrate. Sometimes, the delay due to the warming-up process is intolerable.

The ahead-of-time (AOT) compilation with static shape compilers is also impractical for dynamic shape models in real production. On one hand, it is not practical to enumerate all possible shapes and compile them ahead of time due to the combination explosion of different tensor dimensions, which will cause unacceptable compilation overhead and memory footprint. On the other hand, bucketing the input tensor shapes into a small number of ranges and padding each dimension to the max in each range (i.e., cross-batch padding) will lead to massive redundant computations, which also requires the deployment effort for proper bucketing. An example is that when padding the input shape of BERT [35] model from <9, 33> to <16, 64>, the inference latency of TensorRT [11] is increased to 11.84 ms while the original latency without padding is 4.92 ms, causing 2.41× performance degradation. What is worse, developers do not always know the shape range of the tensors before the execution (e.g., serving online queries of arbitrary lengths of text). When encountering a tensor shape out of the specified range at runtime, the above solution will not work.

To tackle the above problems of static shape compilers, this paper proposes *BladeDISC*, an ML compiler that supports efficient optimization for dynamic shape models. It enables compiling the dynamic shape model once and serving any incoming shapes efficiently. The ML tasks could be per-batch padded (i.e., different batches use different padding), and BladeDISC does not require recompilation when encountering a new shape from the incoming batched samples. It does not need cross-batch padding, avoiding many redundant computations. It makes the following contributions:

► *It is the first work that systematically abstracts, excavates, and represents the hidden shape constraint information of dynamic shape models.* The fundamental challenge of dynamic shape optimization is the lack of tensor shape values at compile-time. We observe that even though there is no exact shape value, there is rich shape constraint information hidden in the tensor computation logic. We systematically abstract the hidden shape information for common graph-level optimizations into two categories. Then we analyze and excavate the shape information from the semantics of each operator in the computation graph, and build global shape constraints of the whole graph (Sec.3.1). Instead of relying on exact tensor shape value for optimizations [14, 30, 91], we design the optimizations based on the excavated global shape information to cope with dynamic shapes. We also propose the cross-level shape representation to address the shape information losing problem across different IR levels and optimization passes (Sec.3.1.3).

► *It proposes the advanced fusion decision approach for dynamic shape models built on shape constraint information rather than exact tensor shape value.* Operator fusion is one of the most important graph transformation optimizations for ML models and data management tasks [14, 27, 30, 32, 41, 63, 73, 86, 91]. The state-of-the-art fusion optimizations rely on the exact tensor shape value for multi-level locality checking (e.g., the locality on registers, the GPU shared memory) to decide whether operators can be fused together. We make an important observation that what the locality checking requires is the equality relationship between tensor shapes of producer and consumer, rather than the shape value itself. With the global shape information, we design the *symbolic dim propagation* approach to propagate and check locality information for dynamic shape fusion decisions (Sec.4.2). We also design the fusion decision pipeline to integrate different fusion decision strategies together for dealing with the increasingly complex graph (Sec.4.1).

► *It proposes the adaptive dynamic shape code generation (codegen) approach with the combination of compile-time and runtime optimizations.* The lack of shape information makes it challenging to generate code that efficiently adapts to various shapes. The codegen *schedule*[2] that is friendly to one shape may perform poorly for other shapes. We combine the compile-time and runtime optimizations together to cope with this issue. At compile-time, we exploit instruction interleaving to help fill hardware pipelines, making the codegen schedule less sensitive to different tensor shapes (Sec.5.1). BladeDISC also compiles for multi-version of code for each operation to enable runtime kernel selection for varied tensor shapes, which shares a similar principle with the alternative kernel selection problems of adaptive query processing [25, 34, 40, 49, 55]. We propose a data-analysis-based heuristic for runtime kernel speculation for different kinds of operations (Sec.5.2).

► It implements a production-ready system to incorporate the optimizations in this paper and provides a comprehensive evaluation of common ML models. The end-to-end optimization pipeline (Sec.6.2), along with the runtime abstraction layer design (Sec.6.1), shows how to build a dynamic shape compiler system with MLIR. We evaluate BladeDISC on a set of commonly used models. The extensive evaluation demonstrates that BladeDISC outperforms PyTorch [64], TorchScript [4], TVM Relay VM (Nimble) [71], ONNX Runtime [10], XLA [14], Inductor [21] (dynamic shape) and TensorRT [11] by up to 6.95×, 6.25×, 4.08×, 2.04×, 2.06×, 7.92×, and 4.16× (3.54×, 3.12×, 1.95×, 1.47×, 1.24×, 2.93×, and 1.46× on average) in terms of end-to-end inference speedup on the A10 and T4 GPU, respectively. BladeDISC has been deployed in a top cloud service provider's AI platform serving massive ML-based data processing and management tasks.

2 BACKGROUND OF DYN-SHAPE COMPILER

2.1 AI Compiler and Limitation on Dyn-Shape

The optimizing compiler approach has been widely used in ML programs in data management tasks. The compiler accepts the machine learning models written in domain-specific languages (e.g., PyTorch [64], TensorFlow [22]), and converts them to the binary on target devices. During the conversion to the target devices, the optimizing compiler is responsible for the microarchitecture-level planning of thread mapping and data locality management (e.g., tiling of MatMul) for codegen.

Modern ML compilers rely on effective operator fusion to better utilize the hardware resource. Note that with the evolution of model structures and the growing ratio of computing power to memory bandwidth, memory-intensive computations have become a rising factor for end-to-end performance [91]. In particular, the rapid computing power growth and the lagging bandwidth follow-up make the performance problems of memory-intensive computation even more pronounced[3]. The state-of-the-art static shape optimizers [11, 91] explore the advanced fusion to leverage different levels of memories for multi-level intermediate data buffering, named *stitch* fusion. The *stitch* fusion helps to reduce off-chip memory traffic and non-computation overhead (e.g., framework-level operator scheduling and device-level kernel launching).

To generate the code after *stitch* fusion, the existing works divide the to-be-fused operators into several groups, generate the code of each group independently, and finally 'stitch' the groups together with the multi-level data buffering in the same kernel. Specifically, AStitch [91] identifies the operators sensitive to parallelism in the fusion as *dominant* ops (e.g., `reduce` op), and groups each dominant op with their non-dominant producers together. It then generates the code of each dominant op and propagates the codegen schedule to other ops within the corresponding group.

[2] As for codegen, *schedule* means how the threads are mapped to hardware to process each data item (e.g., tiling size, on-chip resource management, parallelism configuration).

[3] The computing power of H100 increases by 3.2× compared to A100, and the bandwidth increases by only 1.9×. Moreover, H100 supports FP8 tensor computation and further increases the computing power sharply.

The output of each group is buffered on different levels of memory (i.e., register, shared memory, and global memory) according to the locality between this group and its consumer groups. Finally, the consumer groups will read from their producer groups' output buffers. In AStitch, the locality checking between different groups is done by shape value propagation. For example, when trying to use the GPU shared memory for intermediate data buffering between two operators, AStitch checks how many data items the producer generates and how many data items the consumer requires within each thread block according to static shape values.

The existing static shape ML compilers suffer from either the JIT compilation overhead of each new tensor shape or redundant computations due to large cross-batch padding. There are some efforts to address the dynamic shape problem in ML compilers but with some limitations. Nimble [71] and DISC [94] are proposed to handle dynamic shape models and can apply basic fusion for memory-intensive operators. However, they suffer from insufficient fusion and inefficient codegen optimization for both memory-intensive and compute-intensive computations. Their basic kernel fusion (i.e., fuse element-wise operators into their consumer op) for memory-intensive computations can not help to fully leverage hardware computing power for many modern model structures. Besides the kernel execution itself, the non-computation overhead is another problem caused by insufficient fusion. The massive operators cause massive off-chip memory access and severe non-computation overhead [91]. TensorRT [11] supports the ranged-shape optimization to alleviate this problem but requires users to provide the shape range ahead of time, which cannot work for scenes where the input shape range cannot be obtained before execution.

2.2 Challenges of Dyn-shape Optimization

The absence of shape values makes compiler optimization quite tricky in two aspects.

Graph transformation. One of the most important graph transformations of ML compilers is operator fusion. The decision of the state-of-the-art operator fusion [91] heavily relies on shape information. For example, when trying to fuse reduce[4] operator with its consumer, for which the result of reduce is expected to be buffered on the GPU shared memory, the existing stitch technique requires to guarantee that the number of data items the producer generates and the consumer requires within each thread block are the same according to static shape values. Existing solutions do not support to do stitch fusion decisions without static shape information. There are also some other graph optimizations relying on shape values in existing solutions, like batching GEMMs together when they have the same shape. Losing the shape values makes these graph transformation optimization very tough.

Dynamic-shaped Code Generation. Static shape compilers generate efficient schedules for ML computations according to tensor shapes. As for dynamic shape compilers without shape value at compile time, it becomes challenging for the codegen to adapt to arbitrary tensor shapes efficiently. Besides the inefficient codegen schedule problem, dynamic shape codegen also suffers from the overhead of implicit broadcast. For example, when performing the operation A<?x?> + B<?x?>[5], the compiler should convert it to broadcast(A<?x?>) + broadcast(B<?x?>). This is because the popular frameworks [22, 64] support the implicit broadcast of operators on tensors with different shapes. The broadcast introduces notable performance costs due to the index computations. (Note that integer calculation is very expensive on the GPUs [45].) The broadcast becomes not necessary if the tensor shapes are the same. However, the dynamic shape scene makes it tricky to identify shape equality, resulting in the severe overhead of unnecessary broadcast computations.

[4]The reduce is the operator that reduces an N-D tensor to M-D ($N > M$), like sum of an array, rather than the collective communication op.

[5]A<?x?> means a 2D tensor whose dimension values are unknown at compile-time.

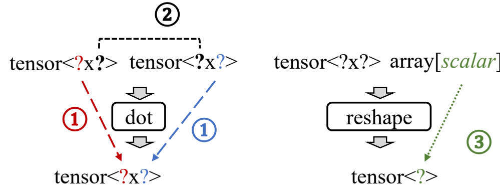


Fig. 1. *Dim equality* analyze: ① input-output infer; ② sibling constraint; ③ shape value extraction.

3 GLOBAL SYMBOLIC SHAPE INFORMATION

The fundamental source of the challenges described in Sec.2 is that, the shape is unknown at compile-time. The compilation is a static action, while shape values are only known at runtime dynamically. Fortunately, we observe that although the accurate shape value cannot be obtained, a large amount of shape constraint information is hidden in the tensor computation graph. By excavating the hidden shape constraints, we can develop graph-level optimizations (e.g., operator fusion in Sec.4) for tensor programs.

In this section, we systematically abstract shape information in a dynamic-shaped tensor computation graph, and thoroughly dig for the hidden shape constraint information for the graph.

3.1 Global Shape Analysis

3.1.1 Dynamic Shape Information Abstraction. Basically, we analyze and excavate the shape information from the semantic of each operator in the computation graph, and build global shape constraints of the whole graph. We abstract the shape information in BladeDISC into two categories: *shape relationship* and *shape property*. A shape consists of a set of dimensions. Thus the shape information can be represented by fine-grained *dim information* (i.e., *dim relationship* and *dim property*). For example, if all the dims are equal for two tensors, the shapes are equal.

Shape relationship. indicates the relationship between different tensors, such as tensor shape equality, equal number of elements between tensors, etc. This information is essential for optimizations like operator fusion.

We observe that there are two most important *dim relationships* for modern machine learning workloads: *dim equality* and *dim collapse equality*. *Dim equality* means that two dims are equal. *Dim collapse equality* means that the computation (e.g., multiply, divide, mod) of a group of dims is equal to another single dim, which is essential to describe tensor shape transformation operations. For example, when we reshape `tensor<?x?>` to `tensor<?>`, the multiplication of the input dims is the same as the output dim.

Shape property. represents the property of a single tensor, such as whether the number of elements of a tensor is able to divide by two. It is useful for optimizations like vectorization.

As for *dim property*, BladeDISC currently analyzes whether a dim is known to be able to divide by constant values (to guide code generation vectorization). It also analyzes and propagates the value range of each dim, which is helpful for code generation speculation in Sec.5.2. With the basic analysis and propagation method in Sec.3.1.2, more dim properties can be easily supported.

3.1.2 Dim Relationship and Property Analysis.

Dim Equality. We analyze the *dim equality* with 3 basic methods:

1. Input-output infer. This is to extract the dim equality between input and output tensors. For example, Fig.1 shows the dim equality between input and output tensors of `dot` operator (i.e., `matmul` op), excavating the equality of M-N dims for matrix multiplication.

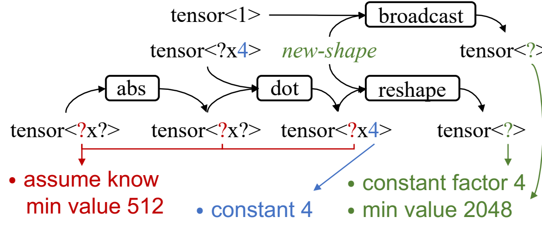


Fig. 2. *Dim property* analyze and propagation.

2. Sibling constraint. This is to extract the dim equality between several input tensors (or output tensors) of an op. For example, the contracting dimensions (i.e., dim K of `matmul`) of the two inputs of `dot` operator have the same dim size, which is shown in Fig.1.

3. Shape value extraction. This is to extract the dim equality according to the specified shape value. For example, some operators require a shape tensor to specify the output shape (e.g., `reshape` operator requires to specify the new shape), for which each element in the shape tensor indicates a dim value of the output tensor. We map the dim value of the output tensor according to each element of the shape tensor. When two dims of different tensors are mapped to the same value, the two dims are equal (`broadcast` and `reshape` in Fig.2). Fig.1 presents a case of `reshape` operator for which the dim of output tensor can be mapped to the shape tensor.

Dim Collapse Equality. BladeDISC identifies *dim collapse equality* according to two ways. The first is to trace integer arithmetic operators (e.g., `mul`, `div`, `mod`) for dim values in the computation graph. The second is to trace shape transformation operators. For example, as for `reshape` operator in Fig.2, the multiplication of input tensor dims is the same as output tensor dim.

Dim Property. We analyze *dim properties* according to known dim properties and algebraic computing properties. For the example shown in Fig.2, when we reshape a tensor of dim `<x,4>` to dim `<y>`, we can indicate that the dim value `y` is able to divide by 4. Assuming we already know that the dim value `x` is greater than 512, we can indicate that `y` will be greater than 2048. We further propagate dim properties of each dim through input-output infer and update dim property when meeting algebraic computing.

3.1.3 Cross-level Shape Representation. The compiler optimization is usually organized as a pass pipeline, where every pass alters the IR of the computation graph sequentially. Although the pipeline manner simplifies the compiler construction, it introduces troubles for shape analysis in two aspects. 1) *Reusability problem.* Suppose there are two passes requiring the shape information for optimization. Since it is difficult to transfer information between passes in existing pipeline design, the most convenient approach is to re-analyze the shape information independently and redundantly with overhead. 2) *Stability problem.* After the processing of some passes, the shape information that is analyzable in previous passes may be lost in later passes as the graph is changed. The instability may lead to conflicts between passes due to different shape information. To address the above problems, we make use of the MLIR's ability to attach information on data types to develop the cross-level IR representation for shape information. Specifically, we bind the symbolic shape information on the data type of each tensor, as is shown in Fig.3. The dimensions of equal size will share the same dim symbol globally on the IR. We also maintain a *dim collapse container* (*DimCollapseContainer* in Fig.3) for *dim collapse equality* information, and maintain a set of *dim symbols* (*SymbolDim* in Fig.3) for *dim properties*.

```
1  func @main(%arg0: tensor<?x?, [@S0, @S1]>, %arg1: tensor<?x4, [@S1, @S2]>, %arg2: tensor<1>) {
2   %0 = abs(%arg0) : (tensor<?x?, [@S0, @S1]>) -> tensor<?x?, [@S0, @S1]>
3   %1 = dot(%0, %arg1) : (tensor<?x?, [@S0, @S1]>, tensor<?x4, [@S1, @S2]>) -> tensor<?x4, [@S0, @S2]>
4   ... ...
5   %2 = reshape(%1, %new_shape) : (tensor<?x4, [@S0, @S2]>, tensor<1>) -> tensor<?, [@S3]>
6   %3 = broadcast(%arg2, %new_shape) : (tensor<1>, tensor<1>) -> tensor<?, [@S3]>
7   return %2, %3: tensor<?, [@S3]>, tensor<?, [@S3]>
8  }
9
10 SymbolDim @S0 …
11 ……
12 SymbolDim @S3 {
13  constant_factors = [4] // can be divided by 4
14  range = [2048,]         // min value is 2048
15 }
16
17 DimCollapseContainer @container {
18  @S3 = mul @S0, @S2
19 }
```

Fig. 3. Pseudo IR after shape optimization and constraint representation for the example in Fig.2.

3.2 Compile-time Broadcast Elimination

With the help of global shape information analysis, BladeDISC can identify many unnecessary implicit broadcasts at compile-time. (An implicit broadcast is unnecessary when the source and destination have the same tensor shape.) By eliminating the unnecessary broadcast, BladeDISC helps to reduce the unnecessary index calculation. Note that integer index calculation on the GPU is very expensive [45]. To further eliminate implicit broadcast operators, BladeDISC presents the multi-version codegen and runtime speculation techniques (Sec.5.2).

4 ADVANCED FUSION DECISION

Operator fusion is one of the most important graph-level optimizations for machine learning jobs. As modern machine learning graphs become increasingly complex, the fusion decision becomes more tricky. Pattern-based methods (e.g., TVM [30]) is difficult to achieve large-grained fusion for complex memory-intensive subgraphs. The memory-intensive oriented optimizer AStitch [91] is not able to fuse compute-intensive operators. What's more, the dynamic shape further increases the difficulty of fusion optimization due to the lack of exact shape value at compile-time, which is essential to determine the fusibility between producer and consumer in existing fusion solutions.

To support different types of fusions given a complex graph, BladeDISC builds up a pipelined fusion decision procedure. The basic insight is to form basic fusions first, and then build up larger fusions by merging the basic fusions. This method is flexible and time-saving. The most complex fusion pass is stitch-fusion for memory-intensive subgraphs, which relies on exact shape value at compile-time for data locality checking in AStitch [91]. We propose the dynamic shape stitch fusion decision approach, with the observation that the locality checking can be done by analyzing shape constraint information rather than exact values.

4.1 Operator Fusion Pipeline

Rather than forming the final fusion at one time, BladeDISC builds up smaller fusions at each pass and merges existing fusions to form larger fusions in the next pass in the pipeline. The pipelined manner breaks the complex decision into a set of smaller jobs, reducing the overall complexity. By reusing existing basic fusion components in previous passes, it saves a lot of fusion exploration

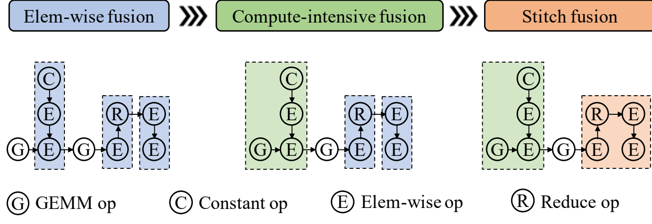


Fig. 4. Fusion decision pipeline in BladeDISC. Ops in the same dotted box will be fused together. The color of the dotted box indicates the fusion type.

time. Meanwhile, the pipeline-based manner is flexible to add new passes for other types of fusion decisions in the future.

Fig.4 describes the fusion decision pipeline of BladeDISC. *Elem-wise fusion* pass is similar to the fusion strategy in XLA [14], which fuses element-wise operators with their consumers together, ends at `reduce` op, and stops by compute-intensive ops. Then the *compute-intensive fusion* pass fuses compute-intensive operators with their pure element-wise consumers together, including `constant` ops. Finally, the *stitch fusion* pass stitches memory-intensive fusions together to form larger fusions (details in Sec.4.2).

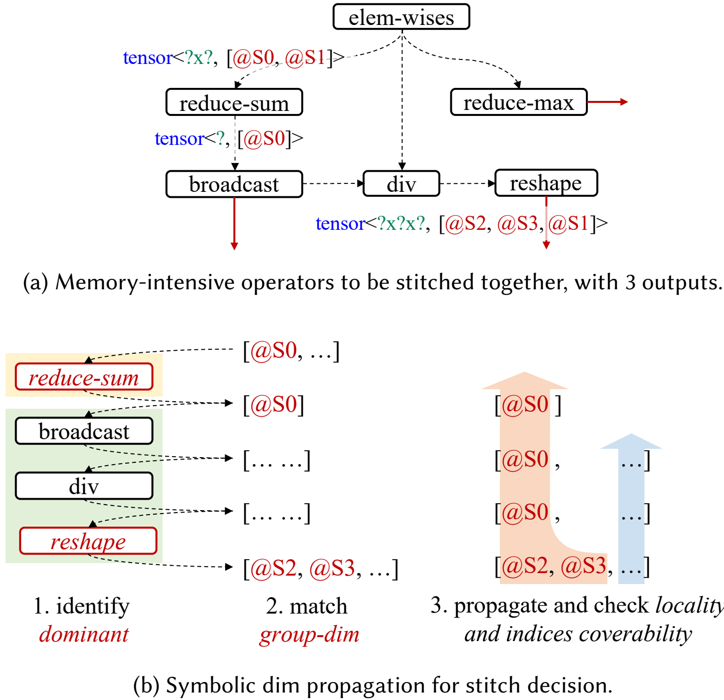
In BladeDISC, it will fuse element-wise operators with compute-intensive ones only if it reduces off-chip memory traffics. Take the example in Fig.4, fusing operator `G` with the following subgraph reduces the off-chip memory write of `G` and the following off-chip memory read of `E`, thus increasing the overall execution efficiency. Note that not all memory-intensive fusions are better to be fused with a producer compute-intensive op. This is because compute-intensive and memory-intensive operators usually require different on-chip resource allocations and thread-level parallelism for good performance. On the one hand, compute-intensive operators rely on large amounts of on-chip resource allocations for good performance, leading to limited parallelism. On the other hand, memory-intensive operators rely on high parallelism for high off-chip memory access efficiency. Force to fuse memory-intensive operators with compute-intensive ones together may result in poor memory access efficiency for memory-intensive operators.

4.2 Stitch-fusion Decision

As described in Sec.2, the existing stitch fusion relies on static shape value for data locality checking. The lack of static shape information makes locality checking and stitch decisions difficult.

We make an important observation that what the locality checking requires is the equality relationship between dims of producer and consumer, rather than the dim value itself. With the global shape information (Sec.3), we design the *symbolic dim propagation* approach for dynamic shape stitch optimization. The basic insight is to propagate along tensor dims with dim equality information for locality checking. Based on the example in Fig.5a, the workflow to make stitch decisions is shown in Fig.5b.

Step 1: Identify *dominant* operators in the to-be-fused subgraph. Note that we borrow the basic insight of the *grouping and then stitching* approach in AStitch [91]. The subgraph will be divided into several operator groups for code generation. The code of each operator group will be first generated independently, and then be stitched together with shared memory. The *dominant* operator is responsible for such a group of operators (called *dominant-group*), covering all its non-dominant producers. As for code generation, BladeDISC would generate the loop structure of each dominant operator independently, and inline its corresponding producers into the loop



(a) Memory-intensive operators to be stitched together, with 3 outputs.

(b) Symbolic dim propagation for stitch decision.

Fig. 5. Stitch optimization for dynamic shape scenario.

structure for code generation (called *input-inline*). The independent loop structure would be finally stitched together through shared memory in the same kernel in BladeDISC. Specifically, the `reduce` operator (`reduce-sum` in Fig.5b) and the output of the subgraph (`reshape` in Fig.5b) are regarded as dominants, just like AStitch [91]. (The `broadcast` is not dominant as its schedule can be inferred from *reshape* with *input-inline*.)

Step 2: Identify the *group-dims* for all dominant operators. We introduce two terms for the technique explanation: *group-dim* and *tile-dim*. Take `reduce` op[6] as an example, supposing which reduces a 2-D tensor (`A<m, n>`) to a 1-D tensor (`B<m>`). On the GPU, BladeDISC will use a tile of threads (usually a thread-block) to reduce the `n` elements and will form `m` such thread tiles. The *tile-dim* is dim `n` here, for which is mapped to a tile of threads. The *group-dim* is dim `m`, for which is mapped to different groups of tiled threads.

As a general rule, BladeDISC will first regard the 'row' dims of `row-reduce` operators as the *group-dims*, and identify the *group-dims* of other non-reduce dominant operators according to *dim collapse equality*. The other dims are *tile-dims* accordingly. The insight is that in the same kernel, all dominants will form the same number of thread tiles, and thus should have the same product of *group-dims*. In Fig.5b, BladeDISC identifies that the first dim of `reduce-sum` is the *group-dim*, whose dim symbol is `@S0`. With shape constraint information, it identifies that `@S0 = @S2 × @S3`. Thus it regards the first two dims of `reshape` as *group-dims*, according to the requirement that all dominants should have the same product for *group-dims*.

[6]We only discuss row-reduce in this paper.

Step 3: Check the locality between *dominant-groups*. It propagates between different *dominant* operators to identify the *group-dims* and *tile-dims* of all passing operators from back to front, starting from each dominant op and ending when reaching another dominant op. The propagation rule is according to the per-element data dependency in each dimension between the output and input tensor for each type of op. For example, for reshape op that reshapes tensor A<x, y, z> to tensor B<a, z>, output-z is mapped to input-z and a is mapped to {x, y}. When the propagation reaches the output tensor of a previous dominant op, it will check whether the newly propagated *group-dims* and *tile-dims* of this tensor are the same as that identified in the first step. If the same, the locality is matched. Otherwise not matched and the stitch decision fails.

Note that we do not see a loop among dominant groups in popular models. If there is a novel model where the dimension relationship cannot be determined due to a loop, the propagation will return and BladeDISC will produce split kernels.

Besides the locality propagation, it also requires checking indices coverability within the dominant-group, if there are outputs that are non-dominant op. This is because all non-dominant operators' codegen schedules are inferred from their corresponding dominant operator (through input-inline). If a non-dominant operator generates a result tensor of the fused kernel, it requires that all the indices of this result tensor should be able to be inferred from its corresponding dominant. Otherwise, the data elements at non-inferred indices will not be generated. For example, a single output tensor of a slice operator is only a part of the input tensor. It means that the indices of a single output tensor of a slice operator are not able to cover all indices of the input tensor. Thus it does not allow the input of slice to be a result of the stitch fusion kernel.

We also expand the connotation of stitch optimization to design the CPU stitch optimization. One difference between GPU and CPU is that the former has an explicitly controlled on-chip memory (i.e., shared memory), while the latter does not have it. The CPU stitch optimization leverages the on-chip cache for efficient intermediate data buffering and reuse. The insight is to increase the temporal locality for intermediate data by rescheduling the computations.

4.3 Compute-intensive Operator Merging

To further increase GEMM efficiency, BladeDISC applies GEMM merging to increase hardware utilization and reduce kernel launch overhead. Specifically, it supports two types of GEMM merging transformation. One is to merge two GEMMs that share the same operand into a single GEMM (i.e., transform A×B and A×C to $A \times [B\ C]$). The other one is to merge two GEMMs with the same shape into a batched GEMM. Note that the dynamic shape scenario makes the latter (i.e., GEMM batching) difficult because it requires that the shape of the GEMMs should be the same. Thanks to the shape constraint analysis technique in Sec.3, BladeDISC can successfully identify shape equality for GEMM batching optimization.

5 DYNAMIC SHAPED CODE GENERATION

We describe how BladeDISC generates efficient code for fused sub-graphs in the dynamic shape scenario in this section. As discussed before, losing shape values makes it challenging to generate code with efficient hardware resource utilization. We design the compile-time and runtime coupled method for shape-adaptive code generation. On the one hand, BladeDISC generates shape-insensitive schedules at compile-time (Sec.5.1). On the other hand, BladeDISC generates multi-version of code for both memory-intensive and compute-intensive computations and selects the best schedule at runtime with a well-designed schedule speculation strategy (Sec.5.2).

```
1 // thread-local reduce of reduce-sum
2 for idx in (threadid : dim-1-of-arg0 step block-size)
3   intermediate = some-elemwise(arg0[n, idx], ...)
4   sum += intermediate
5 // warp-scope reduce of reduce-sum
6 for ...
7   sum += warp_shuffle(sum, ...)
8 ... // cross-warp reduce of reduce-sum
9 // thread-local reduce of reduce-max
10 for idx in (threadid : dim-1-of-arg0 step block-size)
11   intermediate = some-elemwise(arg0[n, idx], ...)
12   max = maximum(max, intermediate)
13 // warp-scope reduce of reduce-max
14 for ...
15   max = maximum(max, warp_shuffle(max, ...))
16 ... // cross-warp reduce of reduce-max
17 ... // the following computations
```

(a) Non-interleaved pseudo code for Fig.5a.

```
1 // thread-local reduce of reduce-sum and reduce-max
2 for idx in (threadid : dim-1-of-arg0 step block-size)
3   intermediate = some-elemwise(arg0[n, idx], ...)
4   sum += intermediate
5   max = maximum(max, intermediate)
6 // warp-scope reduce of reduce-sum and reduce-max
7 for ...
8   sum += warp_shuffle(sum, ...)
9   max = maximum(max, warp_shuffle(max, ...))
10 ... // cross-warp reduce of reduce-sum and reduce-max
11 ... // the following computations
```

(b) Interleaved pseudo code for Fig.5a.

Fig. 6. Pseudo code of instruction interleaving.

5.1 Shape-insensitive Code Generation for Memory-intensive Subgraphs

As for memory-intensive computations, we observe that the most significant performance factor is thread-level parallelism (TLP). The difficulty for dynamic-shape codegen is that it is hard to generate code with optimal TLP without the exact shape values, especially for `reduce` operators. Note that most time-consuming fusions of memory-intensive computations are dominated by `reduce` operators.

Beneath the surface of TLP, the fundamental optimization target is to fill the massive hardware pipelines of GPU architecture. TLP is just the basic programming paradigm to fill the hardware pipelines. Based on this insight, we explore instruction reordering and interleaving in BladeDISC to help fill hardware pipelines, making the code generation schedule less sensitive to TLP and tensor shape values.

There are two main instruction interleaving approaches. One is to interleave the codegen schedule of independent *dominant-groups*. Fig.6 shows an example, where the control flow and instructions of `reduce-sum` and `reduce-max` operators with their inlined producers are interleaved. Before interleaving, each of the two `reduce` operators forms several control-flow structures. The interleaving optimization merges the independent and identical control-flows, reducing control-flow overhead and enabling higher ILP. The other is to improve the software pipeline through instruction interleaving between different loop iterations. BladeDISC unrolls the inner-most loops and reorders instructions to place independent instructions next to each other as much as possible. For example, the thread-local reduce for-loop in Fig.6b (line-2) can be unrolled and the instructions of the unrolled loop can be interleaved to enlarge ILP. Although the loop unroll and interleave is a traditional technique, we find and demonstrate that it is quite effective to make the kernel less sensitive to dynamic-shaped values.

5.2 Multi-codegen and Runtime Speculation.

Besides the shape-insensitive codegen, BladeDISC generates multiple versions of code for each kernel. BladeDISC will select the proper version of the kernel at runtime according to the encountered shape value.

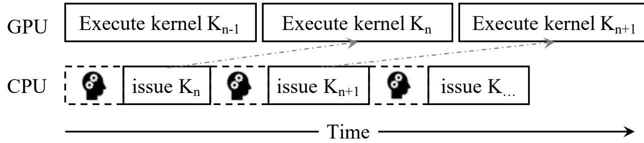


Fig. 7. Overlapped runtime schedule speculation.

Fig.7 shows the basic procedure of kernel execution. CPU speculates the proper schedule and issues the ahead-of-time (AOT) compiled kernel, and GPU executes asynchronously. The schedule speculation is small enough (microseconds) to be overlapped with GPU kernel execution. Moreover, with the shape constraint information, BladeDISC shares the same speculation result for multiple compute-intensive subgraphs with the same shape.

5.2.1 Multi-codegen for Memory-intensive Subgraphs. As for memory-intensive subgraphs, BladeDISC generates two versions of kernels for three aspects of GPU architecture.

1) *Vectorization.* BladeDISC generates a vectorized kernel and a non-vectorized kernel. When it encounters a tensor shape that is divisible by the vectorization factor at runtime, the vectorized kernel will be selected. Otherwise, the non-vectorized kernel will be selected.

2) *Implicit broadcast.* The technique in Sec.3.2 may not be able to eliminate all implicit broadcasts if some of the shape equality cannot be analyzed at compile-time. BladeDISC generates one more kernel that assumes to eliminate all remaining implicit broadcasts. The new kernel will be selected if BladeDISC identifies at runtime that the remaining implicit broadcasts are unnecessary.

3) `Reduce` *operator schedule.* As for `row-reduce` operator, two kernels are generated. One is to process one row with one thread block, the other is with one warp. When the number of rows is small or the number of cols is large at runtime, BladeDISC selects the one-block-one-row kernel for better parallelism and smaller tail latency. Otherwise, the one-warp-one-row kernel is selected to avoid awaits of threads. We use the experiment-based empirical values as thresholds. We do not discuss the schedule details for *reduce* operator because it is not the contribution of this paper.

With the shape information in Sec.3.1.2, BladeDISC can shrink the multi-codegen. For example, if BladeDISC identifies at compile-time that the dim is divisible by 4, it will only generate the vectorized code. This helps to shrink compilation time and memory consumption and avoid the speculation logic at runtime.

5.2.2 Multi-codegen for Compute-intensive Subgraphs. Given a fusion of compute-intensive operator and its following element-wise operators (called *epilogue* of compute-intensive fusion), the performance is dominated by the compute-intensive operator as which usually has much more computations and off-chip memory accesses than the epilogue when fusing together. Thus we can focus on the codegen schedule of compute-intensive operators, while the epilogue follows the schedule of the compute-intensive one.

For compute-intensive operators, different shapes usually require different schedules for good performance. Some works have studied about expanding the shape range each schedule can support efficiently [87]. But a single schedule still cannot serve all shapes. Vendor libraries, like cuBLAS [2], prepare a set of pre-compiled schedules ahead of time and use a rule to switch to different schedules for each given shape. Due to that cuBLAS is not open-sourced, neither can we borrow the schedule selection rule nor can we apply any possible customized compute-intensive fusion with cuBLAS.

To tackle this problem, we design the compile-time multi-codegen and runtime speculation approach for compute-intensive sub-graphs. Given the insight that a range of shapes can share the same schedule for good performance, BladeDISC identifies and compiles the most common schedules

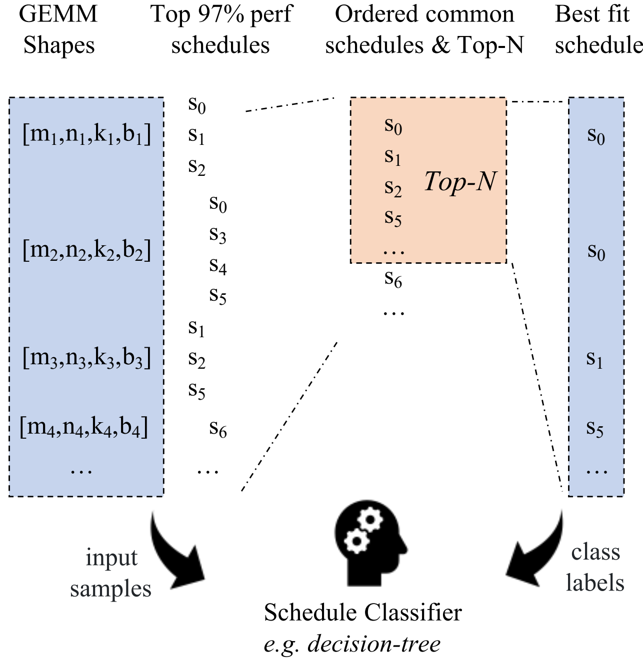


Fig. 8. Schedule selection of GEMM operators.

for different ranges of shapes at compile-time, and speculates the best schedule for the given shape at runtime. Note that the target is different from the recent fast-tuning techniques [82, 93], which are designed to find good schedules at compile-time given specific shapes rather than schedule selection for any given shapes at runtime.

Due to compilation time and binary size limit, it is infeasible to enumerate all possible schedules at compile-time for compute-intensive sub-graph codegen. BladeDISC identifies the most commonly used schedules for various shapes ahead of time. The top-N most common schedules are selected to be compiled for each compute-intensive sub-graph.

Fig.8 shows the schedule selection approach. For each GEMM shape `[M, N, K, batch]`, we profile and identify the schedules that achieve 97% performance of its best schedule we measured. According to the top 97% performance schedules, we get the top-N most commonly used schedules for all analyzed shapes. Finally, we select schedule from the top-N ones for each shape as the *best-fit schedule* according to performance for the shape. Note that the schedule includes the following aspects: block tile, warp tile, pipeline stages, and TensorCore instruction size.

Given the compute-intensive operator shapes and the corresponding *best-fit schedules*, we train a lightweight schedule classifier for runtime schedule speculation. In BladeDISC, we use decision tree [70] as the classifier according to the insight that similar-sized problems tend to share the same schedule.

In this work, we use CUTLASS [3] as schedule profiling tool to get the *best-fit schedule* and build the dataset for schedule classifier training. CUTLASS is demonstrated to be efficient to achieve vendor-library level performance for compute-intensive operators [82]. Sec.7.4.1 describes the details about how we create the dataset to train the schedule classifier to demonstrate the

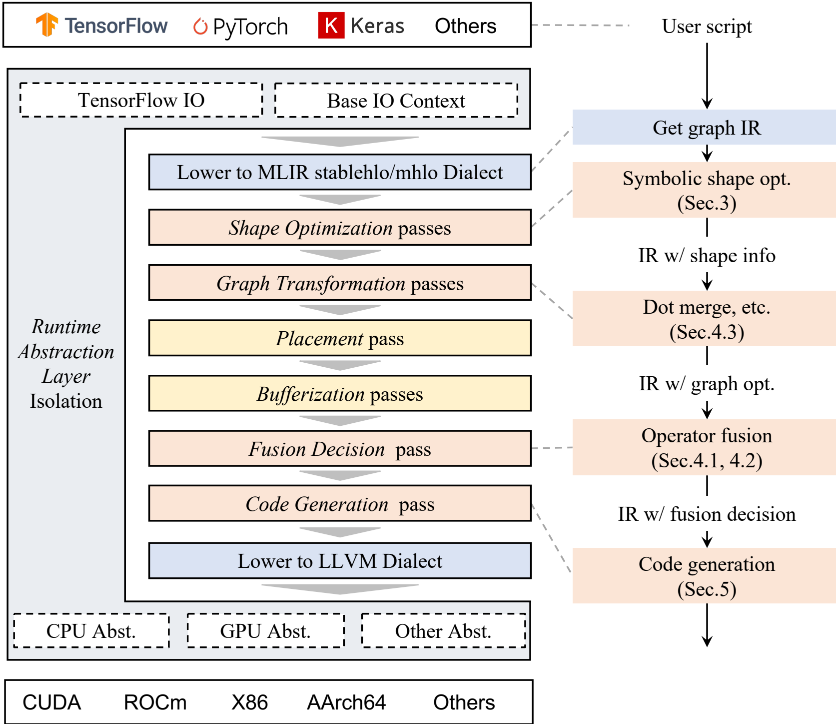


Fig. 9. BladeDISC compiler pass pipeline and corresponding optimization techniques.

effectiveness of this approach. With the basic insight, it can also use other profiling tools [30, 93] to profile and create the dataset. The scale of the dataset could also be customized in practice.

6 SYSTEM DESIGN AND IMPLEMENTATION

We use MLIR as the intermediate representation in BladeDISC. Specifically, BladeDISC makes use of *stablehlo* dialect [1] (or its predecessor *mhlo* dialect [6]) to represent basic ML operators.

6.1 Runtime Abstraction Layer

To isolate the compilation logic from the complex runtime environment of frontends/backends, we design the *runtime abstraction layer* (*RAL*) in BladeDISC. It helps to hide the environment information for the graph optimization and lets the backends focus on their core optimization logic. To avoid the interpretation overhead of existing dynamic shape compilers (e.g., the virtual machine in Nimble [71]), BladeDISC compiles to generate code for both tensor computations on the device side and runtime activities at the host side. Moreover, holistic code generation may enable more opportunities for the co-optimization of compilation and runtime.

Fig.9 shows the function of *RAL* component in the BladeDISC system. The *RAL* component isolates the graph optimization from the interaction with various frontends and backends, by providing the unified interface between tensor computation and the frontends/backends. Specifically, it provides the *TensorFlow IO context* components to interact with TensorFlow/Keras hosts, and *base IO context* to interact with PyTorch and other hosts. As for the backend side, it provides a set of device abstractions for device management, including memory management, task scheduling,

synchronization, and custom calls on popular GPUs and CPUs. The *RAL* design makes it easy to support new frontends and backends by providing the frontend IO context and hardware abstraction implementation, without modifying the core graph optimization logic.

6.2 Optimization Pass Pipeline

Fig.9 shows the optimization pipeline of BladeDISC. The unified interface between BladeDISC and the frontends context is the IR of *stablehlo/mhlo* dialect, through *RAL*. The transformation from TensorFlow to *stablehlo/mhlo* dialect representation is mainly borrowed from the TensorFlow community. The transformation from PyTorch to *stablehlo/mhlo* is implemented by BladeDISC. A PyTorch model has first transformed to TorchScript [4] and goes through *torch-mlir* [13], and finally converts to *stablehlo/mhlo* representation. The unified interface between BladeDISC and the backends context is LLVM IR, through *RAL*. BladeDISC transforms the computation graph represented with *stablehlo/mhlo* dialect to LLVM representation with a set of optimization passes corresponding to the techniques described in this paper. Each of the passes applies a set of optimizations or transformations on the input graph IR and generates the transformed graph IR. The pass-by-pass implementation based on common MLIR dialects (e.g., stablehlo Dialect) results in high reusability from the perspective of both technical principles and implementation.

After lowering the user script written with frameworks like TensorFlow/PyTorch to graph IR (stablehlo/mhlo), BladeDISC first applies the global symbolic shape optimization in the *shape optimization* passes. According to the techniques in Sec.3.1, BladeDISC analyzes global shape information, binds global dim symbols on tensor types, and adds cross-level shape information on IR. With the shape information, BladeDISC applies shape simplification optimizations, including implicit broadcast elimination (Sec.3.2) and constant dimension folding.

After shape optimization, BladeDISC applies a set of graph optimizations in the *graph transformation* passes, including dot merging (Sec.4.3), algebraic simplification, layout transformation (e.g., using `NHWC` format to make full use of TensorCore for `convolution`), etc.

Then the *placement* pass annotates on the IR to place tensor computations on the GPU, if GPU is enabled, and shape calculation logic on the CPU. Note that, unlike the static shape compilers where shape values are constant-folded at compile-time, the dynamic shape compiler needs to infer the shape value at runtime. As BladeDISC compiles for the logic of not only the device side but also the host side, it requires representing the buffer allocation/deallocation explicitly on the host side in IR. The *bufferization* related passes generate allocation/deallocation operators in the IR and transform the usage of tensor values to buffer values (MemRef and related Dialects). Note that the MemRef-based representation generated by bufferization passes presents the memory access behaviors besides the computation itself, which is different from the IR representations in XLA [14] and AStitch [91] where only tensor computations are represented in their IR.

The *fusion decision* pass is followed for the decision of operator fusion. It groups the memory-intensive operators or compute-intensive operators into `fusion` operators according to the techniques described in Sec.4.

Finally, given the graph IR with the fusion decision, the *code generation* pass generates efficient kernels according to the techniques in Sec.5. The computations are transformed into LLVM representation and compiled into binaries of the target architecture.

7 EVALUATION

In this section, we present the detailed evaluation results for BladeDISC. We compare the performance improvement on popular machine learning models with state-of-the-art solutions to show the superiority of BladeDISC.

Model	HuggingFace Name	Type	Configuration
BERT	bert-large-uncased	NLP	24 layers, 1024 hidden-dim, 16 heads, 336M parameters
ALBERT	albert-large-v2	NLP	24 layers, 128 embed-dim, 1024 hidden-dim, 16 heads, 17M parameters
GPT	openai-gpt	NLP	12 layers, 768 hidden-dim, 12 heads, 110M parameters
T5	t5-large	NLP	24 layers, 1024 hidden-dim, 4096 hidden states, 16 heads, 770M parameters
ViT	clip-vit-large-patch14	Vision	Text config: 12 layers, 768 hidden-dim, 12 heads Vision config: 24 layers, 1024 hidden-dim, 16 heads, 14 patch size

Table 1. Information of the benchmark models

7.1 End-to-end Performance

Workloads. We use a set of representative ML models coming from HuggingFace [79] as our evaluation workloads, including BERT [35], ALBERT [50], GPT [66], ViT [36], and T5[67]. Note that HuggingFace is one of the most popular open-sourced model hubs. These models are widely used for language processing and computer vision programs. The detailed information of these models is shown in Table.1. The evaluated batch sizes are 1 and 16. For the language models (i.e., BERT, ALBERT, GPT, and T5), the input sequence length is 64. For the computer vision model (i.e., ViT), the height and width of the input images are both 224, and the number of channels is 3. These input settings are common for industrial scenarios. We enable the auto-mixed-precision (AMP) [59] for the evaluations of all models and all techniques to demonstrate that BladeDISC works well together with AMP.

Baselines. We compare BladeDISC with naive PyTorch framework [64] and a wide range of optimizers including TorchScript [4], TVM Nimble [71], ONNX Runtime [10], XLA [14], PyTorch Inductor [21], and TensorRT [11].

TorchScript integrates nvFuser [16] to support the basic operator fusion on the GPU. As for Nimble, we use both AutoTVM [31] and Ansor [88] to tune the performance with 20,000 steps for each model, as suggested in their documents. We pick the best codegen schedule among the tuning results. We use commit (*15e18*) of TVM [30]. ONNX Runtime [10] converts PyTorch models into ONNX IR and applies various graph optimizations. The version of ONNX Runtime we use is 1.13.1. XLA [14] is a static shape ML compiler that applies operator fusion and many other graph transformations. Note that all the models in our evaluation have equivalent TensorFlow and PyTorch implementations in HuggingFace. Inductor [21] is the native optimizing compiler introduced in PyTorch 2.0 [19]. It automatically converts PyTorch models into Triton [74] codes on the GPU, supporting both static shape and dynamic shape compilation. However, in dynamic shape mode, Inductor cannot effectively excavate the shape relationship between tensors thoroughly and lacks advanced stitch fusion optimization. It also suffers from inefficient codegen without shape value. We present the results of Inductor in both static shape (Inductor-static) and dynamic shape (Inductor-dynamic) modes. TensorRT [11] is provided by NVIDIA to optimize ML models on its GPUs. As mentioned in Sec.2, TensorRT only supports ranged shapes. We observe the abnormal performance of the ranged optimization of TensorRT (Sec.7.1.4) and thus use static shape optimization for the end-to-end comparison. In this way, we can also compare dynamic-shaped BladeDISC with the state-of-the-art static-shaped optimizer. We use TensorRT 8.2.3.

Testbed. We evaluate BladeDISC and the baselines on an NVIDIA A10 GPU and a T4 GPU, respectively. The host-side CPUs are Intel(R) Xeon(R) Platinum 8369B. NVIDIA A10 and T4 GPUs are the most popular NVIDIA accelerators serving inference tasks currently. The PyTorch and TensorFlow versions are 2.1.0 and 2.8.0 in our evaluation. The CUDA toolkit version we use is 11.7, and the cuDNN version is 8.5.0. The operating system is Ubuntu 22.04.

Evaluation metrics. We compare BladeDISC and the baselines by evaluating the model inference (or training) latency of one iteration. We repeat 1,000 times of inferences (or training iterations) after warming up and take the average value of latency. During our test, the accuracy is the same between BladeDISC and other techniques.

7.1.1 Performance on the GPU. Fig.10 shows the performance speedup of BladeDISC over PyTorch and other techniques on the A10 GPU for the five benchmark models. The execution time of PyTorch is normalized to 1. We also present the average speedup of these five models in the right part of the figure. The performance of Nimble on ViT is absent, as Nimble fails to convert the model into Relay IR. Compared with PyTorch, TorchScript, Nimble, ONNX Runtime, and TensorRT, BladeDISC achieves up to 6.95×, 6.25×, 4.08×, 2.04×, and 4.16× speedups, respectively. The average speedups are 3.54×, 3.12×, 1.95×, 1.47×, and 1.46×, respectively.

Fig.10 shows the performance speedup of BladeDISC and other techniques over PyTorch on the A10 and T4 GPU for the five benchmark models with different batch sizes. We also present the average speedup of these five models in the right part of the figure. Some bars are missing in the figure due to the inability of corresponding technologies to optimize the corresponding models. For example, TVM Nimble fails to convert the ViT model into Relay IR. It shows BladeDISC overall outperforms existing optimizers. Compared with PyTorch, TorchScript, TVM Nimble, ONNX Runtime, TensorRT, XLA, Inductor-static, and Inductor-dynamic, BladeDISC achieves up to 6.95×, 6.25×, 4.08×, 2.04×, 4.16×, 2.06×, 2.58×, and 7.92× speedups, respectively. The average speedups are 3.54×, 3.12×, 1.95×, 1.47×, 1.46×, 1.24×, 1.43×, and 2.93×, respectively.

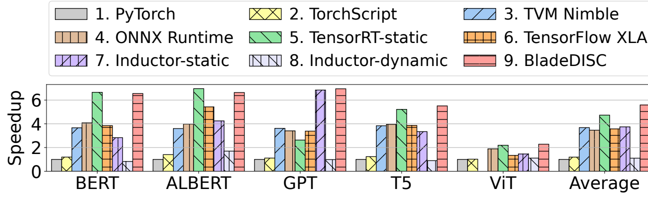
The performance benefit of BladeDISC mainly comes from the effective dynamic shape fusion and shape-adaptive code generation, which helps to reduce off-chip memory access and the non-computation overhead. We present the breakdown analysis in Sec.7.3.

The performance benefit of most optimization techniques with the batch size of 1 is greater than 16. This is because, for smaller batch sizes, memory-intensive operations account for a larger proportion of the end-to-end time. Compared with compute-intensive operations, the optimization potential for memory-intensive operations is larger, which is shown in the breakdown in Sec.7.3.

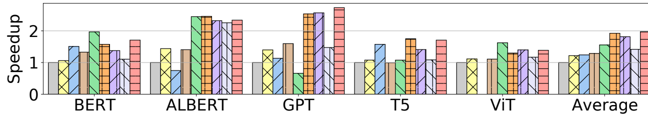
The poor TensorRT performance for the GPT model comes from its inefficient fusion and codegen. The overall execution time (kernel number) of memory-intensive computations for GPT is 699.30us (116) with BladeDISC and 1596.52us (164) with TensorRT. The overall execution time of compute-intensive computations for GPT is 447.71us with BladeDISC and 615.23us with TensorRT. Thus TensorRT has more kernel execution time and more non-computation overhead.

7.1.2 Performance on X86 CPU. We evaluate PyTorch, ONNX Runtime, and BladeDISC on the Intel(R) Xeon(R) Platinum 8369B CPU with one-thread enabled. As the computing power of the CPU is much lower than the GPU, we choose a lighter version of ALBERT [50] (albert-base-v2 from HuggingFace [79], with 11M parameters). Compared with PyTorch and ONNX Runtime, experiments show that BladeDISC exhibits 1.72× and 1.25× speedup, respectively.

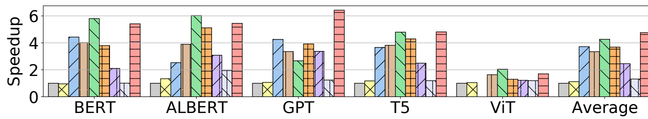
7.1.3 Training Performance. We use DeePMD [77] model and dataset to show the training performance of BladeDISC on an A10 GPU. DeePMD is a popular AI-for-science model to perform molecular dynamics. Experimental results show that compared with TensorFlow, BladeDISC achieves 1.32× and 1.24× speedup on the A10 and T4 GPU, respectively. We also evaluate the training performance of XLA [14] to compare the performance of BladeDISC with static-shaped optimization. Experimental results show that compared with XLA, BladeDISC achieves the speedup of 1.03× on the A10 GPU and 0.98× on the T4 GPU, meaning that the performance of BladeDISC on some training tasks is comparable to or even better than the static compiler.



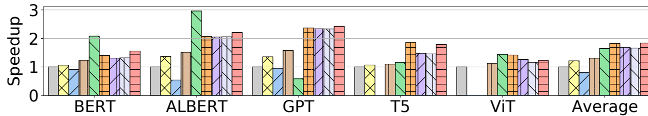
(a) Speed up over PyTorch eager on an A10 GPU with batch size of 1.



(b) Speed up over PyTorch eager on an A10 GPU with batch size of 16.



(c) Speed up over PyTorch eager on a T4 GPU with batch size of 1.



(d) Speed up over PyTorch eager on a T4 GPU with batch size of 16.

Fig. 10. End-to-end Performance of BladeDISC and baselines. The missing bars indicate the corresponding technologies fail to optimize the models in our evaluation.

7.1.4 TensorRT Ranged-Opt Abnormal Performance. We evaluate the performance of TensorRT ranged shape optimization with HuggingFace BERT-large model. We give min-shape, max-shape and optimal shape as <1x16>, <4x256> and <2x64> respectively. It means the batch size ranges from 1 to 4, and sequence length ranges from 16 to 256, with the optimal `<batch-size, seq-length>` as <2x64>. We optimize the model with both the official TensorRT Python API [12] and BladeDISC, and compare the performance of different shapes in the shape range on an A10 GPU [7]. The evaluation results show that the performance of TensorRT ranged-opt is very unstable. Specifically, when batch-size is 1 and sequence-length is 256, TensorRT shows a very severe negative optimization ($-26.4\times$ worse than naive PyTorch baseline), for which the performance of naive PyTorch, TensorRT and BladeDISC are 25.33ms, 668.68ms, and 4.83ms respectively. As the detailed techniques of TensorRT are close sourced, it is hard to figure out the exact reason for the abnormal performance. With Nsight Compute profiling tool [8], we notice that the TensorRT optimized kernels have unusually large launch dimensions, while the output of the model is verified to be correct. From this observation, one possible reason is the unreasonable padding on tensors, due to the insufficient processing of tensors of unknown shapes.

7.2 Comparison with Library-based Solutions

In this section, we compare the performance of BladeDISC with library-based solutions (i.e., FasterTransformer [17] and FlashAttention [33]) on BERT model. Library-based solutions usually target models with fixed structures and require users to rewrite the model relying on the library interfaces. Compared to them, compiler-based solutions have two key advantages. First, compilers can optimize the model automatically without heavy human effort. This is especially beneficial to companies using a large number of models, for which manually rewriting all the models are impractical. Second, compilers exhibit higher flexibility than libraries. For example, if users want to use non-standard models, they cannot use the interfaces provided by FasterTransformer to achieve high performance. By contrast, compilers can optimize them adaptively.

7.2.1 Comparison with FasterTransformer. FasterTransformer is a highly optimized transformer library on the GPU. The per-batch padded seq-length is 64 for the evaluation of this sub-section. We utilize the BERT interface provided by FasterTransformer to evaluate its performance on the A10 GPU. Compared with the HuggingFace baseline, FasterTransformer BERT achieves 6.60× and 1.96 × speedup for batch sizes 1 and 16, respectively. On the other hand, using BladeDISC to automatically optimize the HuggingFace BERT brings 6.55× and 1.71× speedup for batch sizes 1 and 16. This means that BladeDISC can achieve comparable performance to the state-of-the-art manually optimized solution. When enabling the Effective Transformer [15] (a transformer-specific optimization) for the ragged input (i.e., samples have different lengths in the same batch), FasterTransformer achieves 1.98× and 2.36× speedup for batch size 16 when the average seq-length is 48 and 40, respectively. Effective Transformer removes and restores padding before and after attention calculation, which is orthogonal to the fusion and code generation techniques of BladeDISC and could be applied simultaneously in practice. Besides, although FasterTransformer provides the interface of their GPT, T5, and ViT models, we find they do not work for the models we use. This demonstrates that as a library, FasterTransformer suffers from inflexibility.

7.2.2 Combination with FlashAttention. FlashAttention [33] provides a new efficient attention algorithm that reduces the HBM read/write transactions significantly, especially when the sequence length is large. FlashAttention is orthogonal to BladeDISC, as users can replace the attention computation with FlashAttention and then utilize BladeDISC to further optimize the model.

PyTorch 2.0 [19] provides the interface to call the FlashAttention function, which allows the developers to rewrite the model script with this API manually. However, HuggingFace [79] does not rewrite their models to use this interface currently. Hence, we manually replace the attention part of BERT in HuggingFace with the FlashAttention-enabled interface.

To show the advantage of FlashAttention, we use a large sequence length of 512. Experimental results on the A10 GPU show that compared with the original model, enabling FlashAttention optimization brings 1.29× and 1.64× end-to-end speedup for batch sizes of 1 and 16, respectively. On the other hand, using BladeDISC separately brings 3.87× and 1.48× speedup for batch sizes of 1 and 16. By further combining FlashAttention and BladeDISC, we finally achieve 4.72× and 1.92× speedup over the original model for batch sizes of 1 and 16. This is because besides the attention part, BladeDISC further optimizes the memory-intensive computations.

7.3 Breakdown Analysis

7.3.1 End-to-end Performance Breakdown. Fig.11 shows the performance breakdown of PyTorch and the dynamic shape optimizers, i.e., TorchScript, TVM Nimble, ONNX Runtime, Inductor-dynamic, and BladeDISC. The GPU kernels launched during the execution of a model are divided into two categories: memory-intensive and compute-intensive kernels. Fig.11a and Fig.11b show

Subgraph	BladeDISC Latency (us)	BladeDISC #Kernel	Torch Latency (us)
LayerNorm	32.68 (w/o stitch)	3	21.06
	15.70 (w/ stitch)	1	
Softmax	33.40 (w/o stitch)	3	16.05
	12.37(w/ stitch)	1	

Table 2. BladeDISC optimization on memory-intensive subgraphs, given input tensor shape of <1024,1024>.

the execution time proportion of memory-intensive and compute-intensive kernels, respectively. The proportion is figured out by dividing the kernel execution time by the end-to-end inference latency of PyTorch.

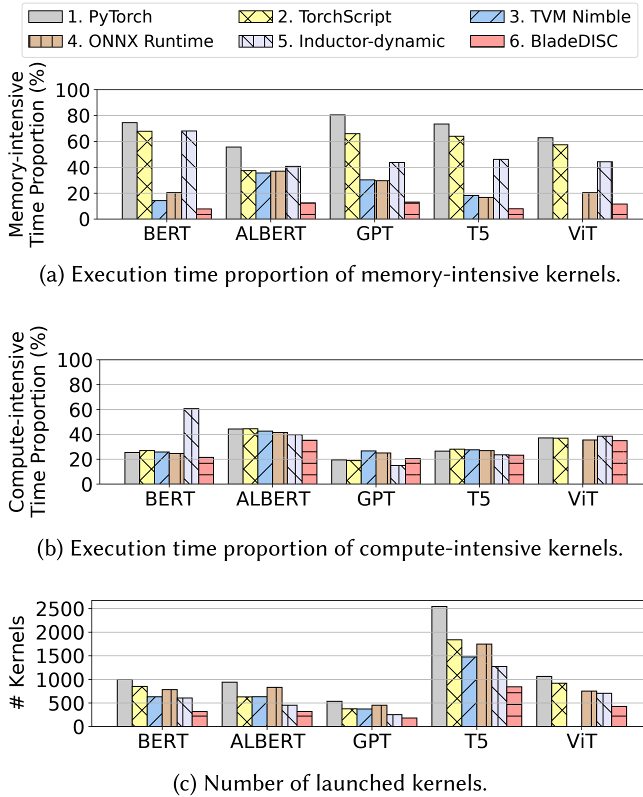
Take BERT as an example. As for the overall memory-intensive computations, BladeDISC significantly boots the execution with speedups of 9.49×, 8.65×, 5.30×, 2.61×, and 8.67× compared with PyTorch, TorchScript, Nimble, ONNX Runtime, and Inductor-dynamic, respectively. The performance benefit mainly comes from stitch fusion optimization (Sec.4) and dynamic shape adaptive code generation (Sec.5). The stitch fusion helps to leverage on-chip memory to buffer intermediate data and reduce off-chip memory access. The dynamic shape adaptive code generation helps to generate more efficient schedules to better utilize the hardware computation resource. As for the overall compute-intensive computations, BladeDISC achieves 1.18×, 1.25×, 1.20×, 1.14×, and 2.82× speedup compared with PyTorch, TorchScript, Nimble, ONNX Runtime, and Inductor-dynamic, respectively. The performance benefit mainly comes from GEMM merging (Sec.4.3) and multi-version code generation (Sec.5.2). Specifically, with the help of GEMM merging, BladeDISC reduces the number of GEMM kernels from 193 in PyTorch, TorchScript, Nimble, and ONNX Runtime to 145. The larger GEMM shapes after merging help to increase hardware utilization. We find that for Inductor-dynamic, the latency of compute-intensive operations is even larger than naïve PyTorch. This is because Inductor-dynamic selects the unsuitable GEMM schedules. Besides the more efficient GPU kernels of both memory-intensive and compute-intensive computations, the efficient fusion in BladeDISC also significantly reduces the kernel numbers by 68.20%, 62.76%, 61.45%, 59.54%, and 47.52% compared with PyTorch, TorchScript, Nimble, ONNX Runtime, and Inductor-dynamic, respectively. The reduced kernel number results in significantly smaller framework-level operator scheduling and hardware-level kernel launching overhead.

7.3.2 Memory-intensive Subgraph Analysis. Table.2 shows the performance of two common memory-intensive subgraphs in modern ML models. It shows that the stitch fusion optimization in BladeDISC significantly boosts the performance of both `LayerNorm` and `Softmax` computations by 2.08× and 2.70× speedup respectively. It also reduces the overall kernel number. Specifically, BladeDISC also outperforms the hand-written PyTorch op implementation of the two subgraphs, by 1.34× and 1.30× speedup respectively.

7.4 Schedule Speculation Analysis

We evaluate the effectiveness of schedule speculation for GEMM schedule selection in this section.

7.4.1 Dataset Creation for Building Decision-tree. To build the decision-tree for GEMM schedule runtime speculation, we measure and create a dataset containing more than 100,000 items of GEMM performance of different shapes under different schedules. We use the CUTLASS [3] profiling tool to measure GEMM performance and create the dataset. Specifically, the M, N, and K dimensions of GEMM shape range from 256 to 2048, and the batch dimension of GEMM shape ranges from 1 to 128, which are common shape ranges of typical GEMM computation in popular ML models. To



(a) Execution time proportion of memory-intensive kernels.

(b) Execution time proportion of compute-intensive kernels.

(c) Number of launched kernels.

Fig. 11. Performance breakdown of PyTorch, TorchScript, TVM Nimble, ONNX Runtime, Inductor-dynamic, and BladeDISC on the GPU. The value in (a) and (b) are figured out by dividing the kernel execution time by the end-to-end inference latency of PyTorch.

guarantee the data diversity, we evenly sample the value in the above shape range of M, N, and K dimensions for which taking the remainder of 256 equals 0, 1, 2, 4, 8, 16, 32, and 64 respectively. This makes the items in the data cover all possible alignments on popular hardware for ML model acceleration. We create the decision-tree classifier on the above dataset according to Sec.5.2.2 for GEMM schedule runtime speculation.

To create the dataset, we only use column-major data layout for the performance evaluation of different schedules given each GEMM shape. We have compared the performance of each schedule under different layouts given a fixed GEMM shape. Extensive evaluations on a large range of GEMM shapes show that, even though different layouts with the same schedule show different performances, the schedule with the best performance under different layouts is usually the same. Thus, data layout is not essential to select the schedule with the best performance.

7.4.2 Schedule Speculation Accuracy. We evaluate the speculation accuracy of the decision-tree classifier. We use 5 schedules as the targets of the classifier. With the top schedule selection approach in Sec.5.2.2, the top-5 schedules are shown in Table.3. It shows that the proportion of occurrences of the top-5 schedules to all original schedules is 95.72%, which shows that the speculation target has high coverage of the possible GEMM schedules. To evaluate the classifier, we use 80% data in the

C_m	C_n	C_k	$stages$	W_m	W_n	W_k	I_m	I_n	I_k	$ratio$
128	128	32	3	2	2	1	16	8	16	68.63%
128	256	32	3	2	4	1	16	8	16	16.08%
128	256	32	2	2	4	1	16	8	8	7.21%
128	128	32	2	2	2	1	16	8	8	2.80%
64	128	32	2	2	2	1	16	8	8	1.00%

Table 3. The top-5 schedules of GEMM classifier targets. $C_{m/n/k}$: the tiling size of block size on M/N/K dimensions. *Stages*: pipeline stages. $W_{m/n/k}$: the warps tiling on M/N/K dimensions. $I_{m/n/k}$: Tensor Core instruction size on M/N/K dimensions. *Ratio*: the proportion of occurrences to all original schedules.

dataset described in Sec.7.4.1 as training data and others as testing data. The decision-tree classifier shows 92% accuracy on the testing data. We collect the ratio of the achieved performance of the predicted schedule to the highest performance of the best schedule of each testing data. It shows that the mean ratio is 99%, and 95.2% GEMM shapes achieve at least 95% of the best performance with the predicted schedule. This means the classifier is effective to get a good schedule for given GEMM shapes for runtime schedule speculation.

We observe that GEMM shapes beyond a threshold usually use a converged schedule, which best matches the hardware characteristics of multi-level memory bandwidth (e.g., global memory, L2 cache, shared memory, and registers) and computing power under enough matrix multiplication computations. In our practice, the GEMM shapes larger than the profiling range during inference usually fall into the converged schedule and result in a good performance. Meanwhile, a too-small GEMM shape usually does not require a perfect schedule as the absolute difference is very small among common schedules. Thus the schedules of the GEMM shapes smaller than the profiling range usually do not significantly impact the end-to-end absolute performance.

8 RELATED WORK

Operator fusion. Operator fusion optimization is extensively studied for data management and ML programs. Tupleware [32], SystemML [27], Weld [63], Tuplex [73], Kasen [86], and FuseME [41] study the fusion optimization on CPU architecture for data management and ML algorithms. They do not study fusion optimization on GPUs, nor do they study dynamic shape optimization for ML programs. Some works study the code generation of compute-intensive operators and support the fusion of element-wise operators with their producer of the compute-intensive operator or consumer of `reduce` operator [26, 30, 37, 75, 88, 93], for which the target is static shape ML models. Some works [61, 89, 90] study holistic operator fusion and pipelined scheduling on heterogeneous systems for pipelined workloads like face detection. Some recent works study the fusion optimization of memory-intensive operators [14, 84, 91, 92] for static shape ML models, especially the stitch fusion optimization [91] leveraging the hierarchical GPU memory for intermediate data buffering. Some works [38, 52, 80] apply fusion optimization in data management tasks like data analytics and query processing on the GPU. None of the above works address the dynamic shape problem for ML programs and will suffer from severe compilation overhead for dynamic shape workloads. There are some works focusing on operator merging of compute-intensive operators [46, 72]. The GEMM batching optimizing in BladeDISC relies on the shape constraint analysis in dynamic scenarios, differing from existing works.

Solutions to dynamic shape models. A practice to alleviate the dynamic shape problem for ML programs is dynamic recompilation [9, 14, 29, 68] for each encountered shape, which suffers from severe compilation overhead and frequent jitter of service time due to the frequent compilation. It

also cannot meet the system expansion requirement due to the long warming-up process. Moreover, it suffers from a huge compilation cache caused by the large shape numbers. The ranged-shape AOT optimization [11] is another approach to alleviate the problem, for which users can provide a range of input tensor shapes rather than a single static shape. However, this approach cannot work for scenes where the input shape range cannot be obtained before execution.

Recently, there are some optimizing compilers supporting dynamic shape ML models. PyTorch Inductor [21], Nimble [71], and DISC [94] are proposed to handle dynamic shape models and apply basic fusion for memory-intensive operators. Instead, BladeDISC proposes the advanced fusion for dynamic shape models. Meanwhile, BladeDISC proposes the compile-time and runtime combined code generation approach that adapts to various shapes efficiently. DietCode [87] focuses on single compute-intensive operator optimization for dynamic shape scenes, rather than memory-intensive operator optimization with the end-to-end flow. IREE [5] focuses on embedded systems and is in the early stage to support server-side devices.

Machine learning systems. Many recent works attempt to improve the ML model performance from different aspects. There are some works proposing efficient runtime systems to serve inference queries. Triton Inference Server [18] and TensorFlow-Serving [62] are widely used serving systems in production. Serverless is also an effective serving option for ML workloads [24, 81]. These works are on top of the model execution engines (e.g. PyTorch [64] and ML compilers) and do not focus on the optimization of ML models execution on microarchitecture, which are orthogonal to BladeDISC. Many works [23, 47, 53, 54, 56–58, 69, 78] are proposed to optimize the large-scale training of ML models. They do not study compilation optimization for dynamic shape ML workloads. ORCA [85] explores token-level batching to optimize transformer-based generative models specifically, which avoids wasted computation by avoiding padding. It does not address the dynamic shape problem of general ML models besides transformer-based generative models. Meanwhile, it does not focus on the fusion and code generation optimization of the per-token computation, for which BladeDISC could be applied on top of the ORCA techniques to optimize the batched tokens from the technical point of view.

Data management and optimizing compiler. The optimization plan switching during runtime is extensively studied for adaptive query processing [25, 34, 40, 43, 49, 55, 83], to which BladeDISC shares a similar principle of multi-kernel/plan generation and switching. BladeDISC studies the new problem of ML programs and generates and selects the multi-kernel with a set of new methods. Leveraging compilation technologies to optimize queries is an enduring topic in data management areas [39, 40, 49, 60, 65]. LingoDB [48] utilizes MLIR [51] to build a layered query compilation stack. BladeDISC also uses MLIR to build the compilation pass pipeline. Instead of database queries, BladeDISC focuses on compilation optimizations for ML models.

9 CONCLUSION

In this paper, we design and implement an industrial-grade dynamic shape compiler system, named BladeDISC. To tackle the fundamental challenge of the absence of shape values, we systematically abstract and excavate the shape information and design a cross-level symbolic shape representation. With the insight that what fusion decision relies upon is the shape relationship, we propose the dynamic shape fusion approach with shape information propagation. We design the compile-time and runtime combined approach to generate efficient code that adapts to arbitrary tensor shapes. Results show that BladeDISC outperforms PyTorch, TorchScript, TVM, ONNX Runtime, XLA, Torch Inductor (dynamic shape), and TensorRT by up to 6.95×, 6.25×, 4.08×, 2.04×, 2.06×, 7.92×, and 4.16× (3.54×, 3.12×, 1.95×, 1.47×, 1.24×, 2.93×, and 1.46× on average), respectively.

REFERENCES

[1] Cited April 2023. Stablehlo, backward compatible ML compute opset inspired by HLO/MHLO. https://github.com/openxla/stablehlo.
[2] Cited January 2023. Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas.
[3] Cited January 2023. CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass.
[4] Cited January 2023. Introduction to TorchScript. https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html.
[5] Cited January 2023. IREE. https://github.com/google/iree.
[6] Cited January 2023. MLIR-HLO: A Standalone "HLO" MLIR-based Compiler. https://github.com/tensorflow/mlir-hlo.
[7] Cited January 2023. NVIDIA A10 GPU Accelerator. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a10/pdf/A10-Product-Brief.pdf.
[8] Cited January 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.
[9] Cited January 2023. NVIDIA TensorFlow User Guide. https://docs.nvidia.com/deeplearning/frameworks/tensorflow-user-guide/index.html.
[10] Cited January 2023. ONNX Runtime. https://onnxruntime.ai.
[11] Cited January 2023. TensorRT. https://developer.nvidia.com/tensorrt.
[12] Cited January 2023. TensorRT Python API Reference. https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/.
[13] Cited January 2023. The Torch-MLIR Project. https://github.com/llvm/torch-mlir.
[14] Cited January 2023. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.
[15] Cited June 2023. Effective Transformer. https://github.com/bytedance/effective_transformer.
[16] Cited March 2023. Introducing nvFuser, a deep learning compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/.
[17] Cited March 2023. NVIDIA FasterTransformer. https://github.com/NVIDIA/FasterTransformer.
[18] Cited March 2023. NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.
[19] Cited March 2023. PyTorch 2.0 Release. https://pytorch.org/blog/pytorch-2.0-release/.
[20] Cited March 2023. TensorRT Dynamic Shape. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#work_dynamic_shapes.
[21] Cited March 2023. TorchInductor: a PyTorch-native Compiler with Define-by-Run IR and Symbolic Shapes. https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747.
[22] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283.
[23] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. Accelerating Recommendation System Training by Leveraging Popular Choices. *Proc. VLDB Endow.* 15, 1 (2021), 127–140.
[24] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. VLDB Endow.* 15, 10 (2022), 2071–2084.
[25] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 261–272.
[26] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205. https://doi.org/10.1109/CGO.2019.8661197
[27] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 11, 12 (2018), 1755–1768.
[28] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 7, 7 (2014), 553–564.
[29] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.* 37, 3 (2014), 52–62.

[30] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594.

[31] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3393–3404. https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html

[32] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.* 8, 12 (2015), 1466–1477.

[33] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *CoRR* abs/2205.14135 (2022). https://doi.org/10.48550/arXiv.2205.14135 arXiv:2205.14135

[34] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Foundations and Trends® in Databases* 1, 1, 1–140.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186.

[36] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.* OpenReview.net.

[37] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2021. ETO: Accelerating Optimization of DNN Operators by High-Performance Tensor Program Reuse. *Proc. VLDB Endow.* 15, 2 (2021), 183–195.

[38] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1603–1618.

[39] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient stream processing through adaptive query compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2487–2503.

[40] Tim Gubner and Peter Boncz. 2022. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. *Proceedings of the VLDB Endowment* 16, 4 (2022), 829–841.

[41] Donghyoung Han, Jongwuk Lee, and Min-Soo Kim. 2022. FuseME: Distributed Matrix Computation Engine based on Cuboid-based Fused Operator and Plan Generation. In *Proceedings of the 2022 International Conference on Management of Data.* 1891–1904.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016.* IEEE Computer Society, 770–778.

[43] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *Proceedings of the 2022 International Conference on Management of Data.* 1360–1374.

[44] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. 2015. Resource Elasticity for Large-Scale Machine Learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 137–152.

[45] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *CoRR* abs/1903.07486 (2019). arXiv:1903.07486 http://arxiv.org/abs/1903.07486

[46] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 47–62.

[47] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2021. COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy

Compression. *Proceedings of the VLDB Endowment* 15, 4 (2021), 886–899.

[48] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2389–2401.

[49] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 197–208.

[50] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[51] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14.

[52] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.

[53] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018.

[54] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. 2022. Harmony: Overcoming the hurdles of GPU memory capacity to train massive DNN models on commodity servers. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2747–2760.

[55] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 659–670.

[56] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2262–2270.

[57] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: A Graph-based System Approach to Scaling Large Embedding Model Training. In *Proceedings of the 2022 International Conference on Management of Data*. 470–480.

[58] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proceedings of the VLDB Endowment* 16, 3 (2022), 470–479.

[59] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[60] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[61] Chanyoung Oh, Zhen Zheng, Xipeng Shen, Jidong Zhai, and Youngmin Yi. 2020. GOPipe: A Granularity-Oblivious Programming Framework for Pipelined Stencil Executions on GPU. In *PACT '20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020*, Vivek Sarkar and Hyesoon Kim (Eds.). ACM, 43–54.

[62] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR* abs/1712.06139 (2017). arXiv:1712.06139 http://arxiv.org/abs/1712.06139

[63] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035.

[65] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proceedings of the VLDB Endowment* 14, 2 (2020), 202–214.

[66] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[67] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[68] Jonathan Raiman. 2018. Dali: Lazy Compilation of Dynamic Computation Graphs. In *Workshop on Systems for Machine Learning and Open Source Software at NeurIPS 2018*.

[69] Alexander Renz-Wieland, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. 2022. NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. In *Proceedings of the 2022 International Conference on Management of Data*. 481–495.

[70] S. Rasoul Safavian and David A. Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE Trans. Syst. Man Cybern.* 21, 3 (1991), 660–674.

[71] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems* 3 (2021), 208–222.

[72] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 909–923.

[73] Leonhard F. Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1718–1731.

[74] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[75] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).

[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[77] Han Wang, Linfeng Zhang, Jiequn Han, and Weinan E. 2018. DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics. *Comput. Phys. Commun.* 228 (2018), 178–184. https://doi.org/10.1016/j.cpc.2018.03.016

[78] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *Proceedings of the 2022 International Conference on Management of Data*. 1301–1315.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, 38–45.

[80] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 107–118.

[81] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2022. Serverless data science-are we there yet? a case study of model serving. In *Proceedings of the 2022 International Conference on Management of Data*. 1866–1875.

[82] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org.

[83] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. 931–944.

[84] Xiaodong Yi, Shiwei Zhang, Lansong Diao, Chuan Wu, Zhen Zheng, Shiqing Fan, Siyu Wang, Jun Yang, and Wei Lin. 2022. Optimizing DNN Compilation for Distributed Training With Joint OP and Tensor Fusion. *IEEE Trans. Parallel Distributed Syst.* 33, 12 (2022), 4694–4706.

[85] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[86] Mingxing Zhang, Yongwei Wu, Kang Chen, Teng Ma, and Weimin Zheng. 2016. Measuring and Optimizing Distributed Array Programs. *Proc. VLDB Endow.* 9, 12 (2016), 912–923.

[87] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic Optimization for Dynamic Tensor Programs. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org.

[88] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879.

[89] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: a versatile programming framework for pipelined computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, Hillery C. Hunter, Jaime Moreno, Joel S. Emer, and Daniel Sánchez (Eds.). ACM, 587–599.

[90] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2019. HiWayLib: A Software Framework for Enabling High Performance Communications for Heterogeneous Pipeline Computations. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 153–166.

[91] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 359–373.

[92] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924* (2020).

[93] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 233–248.

[94] Kai Zhu, Wenyi Zhao, Zhen Zheng, Tianyou Guo, Pengzhan Zhao, Junjie Bai, Jun Yang, Xiaoyong Liu, Lansong Diao, and Wei Lin. 2021. DISC: A Dynamic Shape Compiler for Machine Learning Workloads. In *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*, Eiko Yoneki and Paul Patras (Eds.). ACM, 89–95.

Received January 2023; revised April 2023; accepted May 2023